

Graphic language translation with a language independent processor

by RONALD A. MORRISON

General Electric Company
Cincinnati, Ohio

INTRODUCTION

Since January 1963 when "Sketch Pad, A Man-Machine Graphical Communication System"¹ by Ivan E. Sutherland was published as a Ph.D. Thesis at M.I.T., graphic displays on computers have been used for a variety of experimental^{2,3} and production⁴ purposes. The motivation behind the system described here, is to provide an interactive graphic input/output facility to a number of *existing* software systems. Our business, like many others, has developed over the past 10 years a number of sophisticated and complex "Computer Aided Design" systems. These systems include principally numerical control part programming systems and engineering design analysis systems. Although these systems have proven themselves to be a *necessary* part of the design, and manufacturing process, nevertheless preparing input and assimilating output is still a time consuming process for the users. The interactive display speeds up these processes. Its output is more meaningful, too, since pictures as well as numbers and text are used as communication media between the man and the machine.

The term "graphic language" has been used ambiguously, in the literature, to describe at least three different types of language used in graphic processing.

1. The input stream is in the form of actions taken by a console operator.
 - a. draw with light pen
 - b. type names and numbers
 - c. push buttons
 - d. light pen references of objects on the screenA language translator translates these actions into invocations of appropriate procedures. These procedures perform requested actions and provide displayed feed back to the user.
2. Input is in the form of pictures existing on film or other media. In this case the language trans-

lator is a pattern recognizer which recognizes and extracts meaning from these pictures.⁵

3. A set of programming tools (functions and subroutines) are embedded in a "host" language (e.g. FORTRAN). Using these tools lightens the load of the programmer of the graphic system.

The primary concern of this paper is the language and corresponding processor discussed above as type one. A set of service functions and subroutines (type three above) are also embodied in the system but are of secondary importance.

The system described here borrows freely from the best of earlier graphics work. It is set in an environment similar to GRAPHICS I² (i.e., a remote display with local compute power communicating with a central multiprogrammed computer). It uses a list structured data base, as expounded by Ling,⁶ for storing both information about and relationships between graphic entities. This is similar to the "plex" structure used by D. Ross in the AED³ system. This paper purports to contribute to the state-of-the-art by defining in Backus normal form⁷ the class of language into which graphic statements may be imbedded, and then defining a language processing scheme that will translate any statement of this class. A useful feature of this scheme is that new graphic statements can be composed at the graphic display causing driving tables for the language processor to be produced automatically. Thus the language is automatically extendible within the class described below.

Graphic language

The class of language under scrutiny is a simple phrase structured grammar.⁸ Statements of the language are of the following form:

Major Word <part>< > ,

Minor Word <part>< > *Minor Word* etc.

The following are examples of some statements in the language:

Line <point definition> <point definition>
Joint <joint definition> ,
 Radial Deflection <decimal number>

The language is defined in general terms as follows:

```
<Graphic Program> ::=
    <Graphic Statement>|
    <Graphic Program> <Graphic Statement>
<Graphic Statement> ::=
    <Terminal Symbol> <Trailer>|
    <Graphic Statement> <Terminal Symbol>
    <Trailer>
<Trailer> ::= <Part>|<Trailer> <Part>|<Null>
```

The elements of the terminal vocabulary (denoted by <Terminal Symbol>) must be defined for the specific application. The portion of the non-terminal vocabulary that was left undefined <part> depends for definition upon the application and the specific graphic hardware. It consists of such things as <decimal numbers>, <light pen coordinate pairs> or <display item reference lists>.

Appendix A contains the Backus normal form language description of the graphic language designed for a large engineering design analysis system called "Multishell." Note that in this application members and joints are graphically just lines and points respectively. They may be defined as simply as:

1. Push "Position" button while pointing light pen at desired position on the screen
2. Push "Send" button

in which case the line is drawn from the last defined point to the new light pen position, or as laboriously as:

1. Point light pen at "Draw Member" menu item
2. Type x, y coordinates of start point
3. Type x, y coordinates of end point
4. Push send button

Note also that statement types (e.g. "Draw Member") are modal (i.e., do not have to be respecified before each subsequent statement).

Accepting the premise that any graphic statement will be an element of the class described above, we now attend to defining a language processing scheme that will translate these statements and compile useful code (display file) as a result.

Language processor

While it is, in principle, possible to let the input stream directly invoke the necessary procedures, it is convenient to interpose a lexical analyzer⁹ between the graphic input devices and the procedures themselves. The duties of this lexical analyzer include

isolating identifiers, literals, and the operators and delimiters of the graphic language. More important, however, it handles the switching of displayed menus as the user makes his selections with the light pen, and certain graphic manipulations (e.g., light pen tracking, picture moving, scaling, etc.). It also does some primitive syntax checking.

Since menu switching is one of the more significant duties performed by the lexical analyzer, it is embellished here. In appendix A, the "Case Identification Statement" contains the terminal symbol "Multishell" followed by a number of "parts." This statement is implemented as follows:

1. The following menu is displayed on the screen:

```
SELECT A PROGRAM
ADAM
APT
CYCLE
DYNARS
MULTISHELL
```

2. The console user selects MULTISHELL by pointing the light pen at it.
3. This action by the user causes the lexical analyzer to delete that menu.
4. The terminal symbol *Multishell* is deposited in the attention file.
5. The following new menu is displayed:


```
CASE IDENTIFICATION
GO TO TYPEWRITER FOR
FURTHER INSTRUCTIONS
```
6. The typewriter types


```
IDENTIFICATION OF THIS CASE *
```
7. User types the identification code.
8. System user interaction continues until all "parts" of the case identification statement are supplied.

The language processor is modularized into two major subsystems. The first subsystem, *the lexical analyzer*, deals with the input language from the graphic console to a very shallow level. It is concerned only with the words or vocabulary of the language with little concern for the structure (syntax) and meaning (semantics) of the language. It recognizes inputs from the function keys, typewriter, and light pen, sorts them out and, where necessary, concatenates them into identifiers, real and integer literals, and the terminal symbols of the language.

The second subsystem, *the statement subroutines*, is concerned with the syntax and semantics of the graphic language. A statement subroutine exists for each statement in the language. It performs the actions indicated by the semantics of the statement. Many of the statement subroutines perform similar actions (i.e., manipulate or add to the data base, and add items

to the display file). Thus the library of subroutines may conveniently be divided into two classes (i.e., the statement subroutines themselves and the service routines and functions used in common by many of the subroutines). The statement subroutine may be thought of as the executive routine for that specific statement. In terms of language translation it performs syntactic analysis, code generation (display file), and the maintenance of a list structured data base. The service routines are a set of tools used by the statement subroutine in performing its functions. They provide a discipline towards generalization. As functions are found to be of common usage they are added to the service package. A block diagram of the system is illustrated by Figure 1.

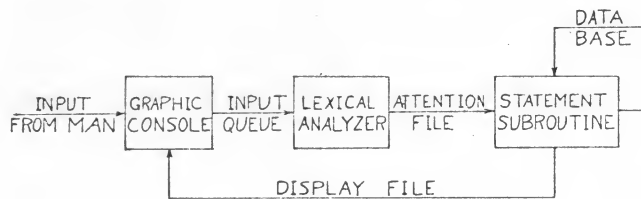


Figure 1 - System block diagram

The system is a simple feedback system where the man's input perturbs the system to a change in state which is reflected by a change in the display. The state of the machine is kept in the data base and the display file. The data base is list structured so that relationships that exist between displayed entities may be conveniently recorded.

The specific formats of all the data structures (i.e., the input queue attention file, data base, and display file) are machine dependent. However, their contents and logical structure are outlined here.

Input queue

The input queue receives its input directly from the graphic console. This is the input stream of characters to the entire system. There are several types of characters; function keys, alphanumeric keys, pen positions, and references to objects "seen" by the pen. The input stream is typically formatted one character per computer word with an identification field in each word to indicate which type of character it represents. In the interest of clarity and consistency an example will be introduced at this point and carried on through the description of all the data structures. The user "draws" a line on the console by the following actions, assuming the system has already been placed in the draw line mode:

1. typing 1.5, typing comma, typing 2.75,
2. positioning the pen at the line end point, and pushing the "position" button,
3. pushing the statement terminating "send" button.

This results in the input queue shown in Table I.

TABLE I - Input queue

Identification	Data
Alphanumeric	1
Alphanumeric	.
Alphanumeric	5
Alphanumeric	,
Alphanumeric	2
Alphanumeric	.
Alphanumeric	7
Alphanumeric	5
Function Key	Position
Pen Position	X Y
Function Key	Send

Attention file

The attention file is the output of the lexical analyzer. It is in a form that is convenient for the statement subroutines. The alphanumeric characters are concatenated into real numbers, integers, or symbols. The attention file, that results from lexical analysis of the input queue example above is illustrated by Table II.

TABLE II - Attention file

Type	Data
Terminal symbol	Line
Terminal symbol	X Y Coordinate pair
Real number	1.5
Real number	2.75
Terminal symbol	Position
Position	X Y
Terminal symbol	Statement Terminator

Data base

The data base is a list structured representation of the state of the graphic machine at any moment. The structure is hierarchical in that an item has either higher, lower, or the same level as any other item. Any number of relationships between items may exist.

The data base for the example statement is shown in Figure 2. The rectangular boxes represent items and the circular ones represent relationships and are called conjunctions.⁶ The rings represent the list links.

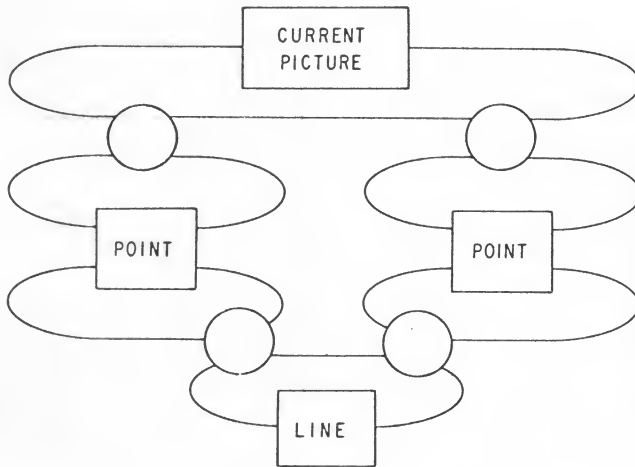


Figure 2—Data base

Display file

The display file is the program which drives the graphic display. It contains such commands as draw line, draw point, or print character. The display file that results from our example is shown in Table III.

TABLE III—Display file

Identifier	Data	
Position Beam to	X	Y
Draw Point		
Draw Line to	X	Y
Draw Point		

This is the final display file entry that results from our example. The lexical analyzer, while processing our input, forms some temporary display file entries e.g., the alphanumeric characters, points, etc., but these are subsequently removed when the statement subroutine completes its job.

Lexical analyzer

The lexical analyzer has seven major duties.

1. Concatenates literals (real, integer) and symbols.
2. Recognizes terminal symbols. Identifies them as such so that the proper statement subroutine may subsequently be called.

3. Recognizes statement termination and as a result causes the appropriate statement subroutine to be called.
4. Maintains a temporary display file for editing purposes (points, characters, operator language guides).
5. Produces as output the attention file (AF) to be passed along to the statement subroutines.
6. Displays new menus as selections are made.
7. Does some primitive syntax checking.

The fact that the lexical analyzer is required to know very little about the meaning of the language it processes, allows it to be written very generally. It can, in fact, be practically language independent. This means that adding new language requires no change in the lexical analyzer itself. Instead appropriate additions are made to its driving tables. Figure 3 shows the lexical analyzer, its associated tables, and how it fits in with the rest of the system.

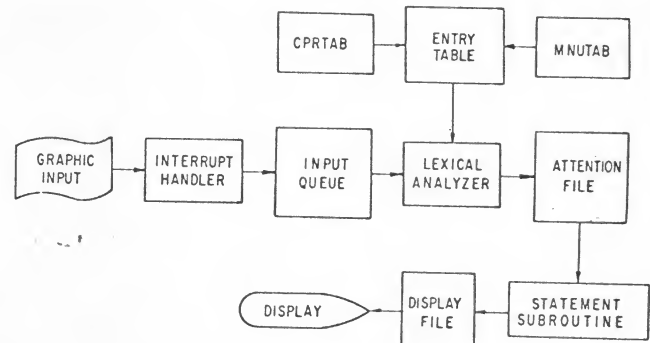


Figure 3—Lexical analyzer

The lexical analyzers operation is described as follows. The lexical analyzer operates between two ring buffers. It gets its input stream of characters from the *Input Queue*. It is driven by the *Entry Table* which contains the entries that the lexical analyzer must make in its output files as a result of each possible input. The specific entry in the *Entry Table* is pointed to by either the *Character Property Table* for function keys or alphanumeric keys or by the *Menu Table* for menu references. The output files that are driven by the lexical analyzer are the *Attention File* and the *Display File*. The *Attention File* is the other ring buffer mentioned above and passes the lexically analyzed information on to the *Statement Subroutines*. Some statements also cause output to be put into the *Display File* (i.e., changes in display of menus or immediate reactions to user actions).

The contents and logical structure of the *Entry Table* are illustrated below by Table IV.

TABLE IV—Entry table

No. of Phrases in this statement Increment added if this is a phrase No. of parts in this phrase	Header
No. of words for Attention File No. of words for Display File	
Enable teletype flag Enable light pen flag	Header
Attention File Entries	
Display File Entries	

The Entry Table is the lexical analyzers key driving table. The first three words allow the lexical analyzer to do its primitive syntax check. This consists of checking to see that the statement has the correct number of phrases and that each phrase has the correct number of parts. The entries are moved by the lexical analyzer to the appropriate output file.

The logic of the Lexical Analyzer is illustrated by Figure 4.

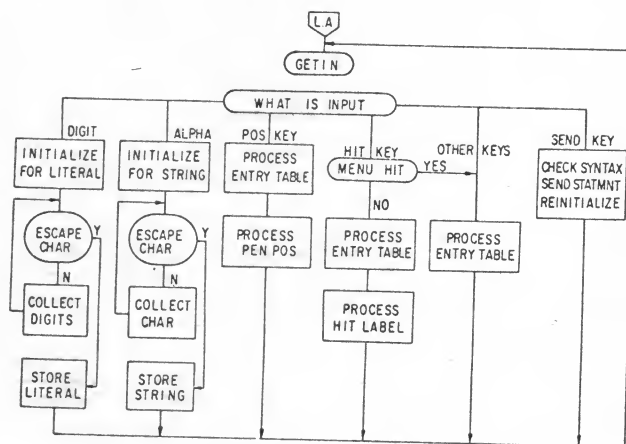


Figure 4—Lexical analyzer logic

Statement subroutines

The statement subroutines consist of the application routines and a package of general purpose subroutines. A detailed description of all statements in a specific application is outside the scope of this paper. However, a flow diagram of the skeleton of a statement subroutine is illustrated by Figure 5.

The general purpose service subroutines fall into two classes:

1. The list processing routines are used for creating, manipulating (e.g., searching, relating, etc.) and destroying lists.
2. The display file processing routines are used to display items and remove items from the display.

Environment

The proposed environment of the system is described as follows. A small digital computer is placed at the end of the remote phone line with the graphic console. It, as well as other remote terminals (e.g., teletypes and high speed printer, card reader combinations) communicate with the GE 635 central computer through voice grade lines at 2.0 kb rate. The GE 635, operating under the GE Comprehensive Operating Supervisor and the GE Remote Terminal Supervisor, provides a multiprogrammed remote terminal environment for the system. The interface between the main frame and the remote processor is now at the attention file and the display file. That is the input queue, lexical analyzer, attention file and display file are located in the remote processor. The statement subroutines, data base, and attention file are located in the main frame. With this division of responsibility, the remote processor does the tasks that require only a shallow understanding of the problem i.e., the lexical analysis. The main frame contains the data base and statement subroutines and does the more detailed processing. This system is pictured in Figure 6.

Application flexibility

One important practical consideration that has not yet been mentioned is the question of application flexibility. The lexical analyzer embodies a flexible table driven translation methodology. However, coding these tables by hand for every menu of every application would be a tremendously tedious task. Therefore, a convenient means of building these tables needs to be provided.

Ideally, the application programmer would like to sit at the graphic display and compose his menus. But, provision must also be made for labeling items in a menu, cross referencing them to other menus, and indicating the function code by which the statement subroutine will recognize the item in the attention file. A sub language is proposed which uses the lexical analyzer itself to generate new menus. Of course, the tables for this sub language must be coded by hand. But, once the system is thus bootstrapped, new applications can be added easily.

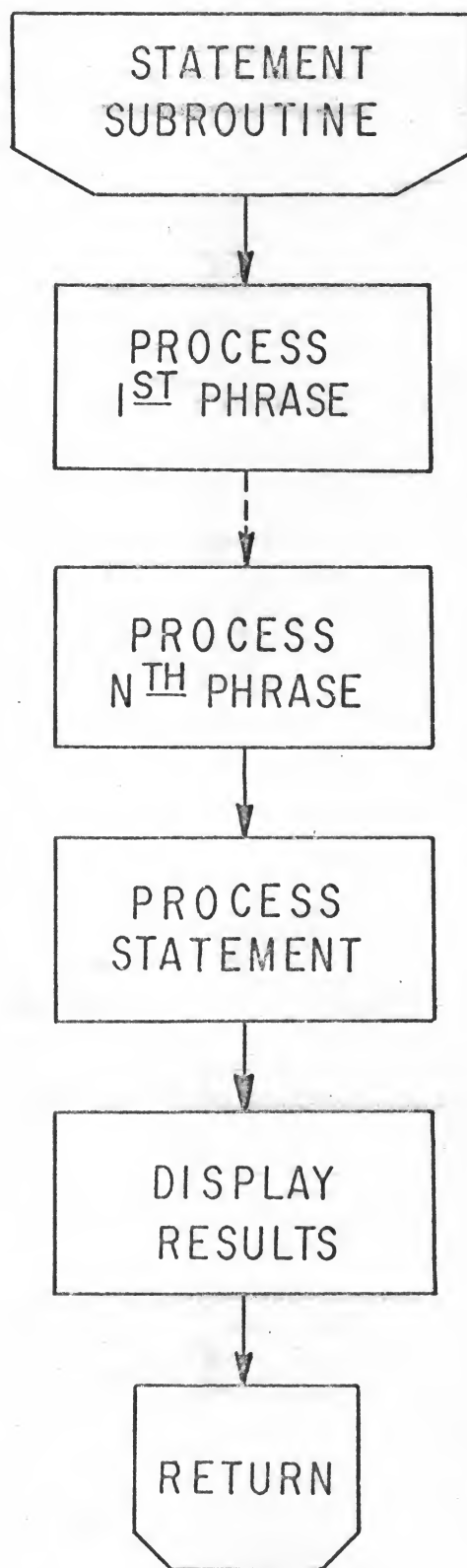


Figure 5 — Statement subroutines

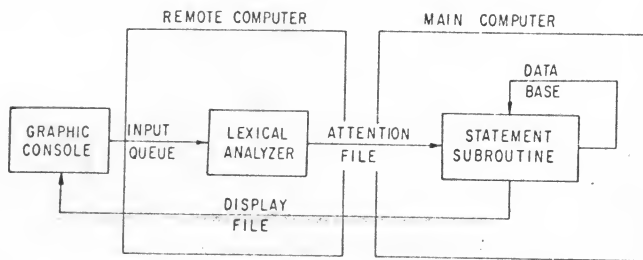


Figure 6—System Environment

SUMMARY

The major premise on which this paper is based is that all graphic statements fall into the class described earlier. If the potential user of the system will accept this restriction, his application can be implemented with a minimum of effort. He must write the statement subroutines, which he would have to do in any case. Then he must compose the language by which he will communicate with these subroutines. He does this graphically at the graphic console. The software system does not change from application to application except for the statement subroutines and the contents of the lexical analyzer's driving tables.

Both D. Ross with AED³ and L. Roberts with extensions to VITAL¹⁰ have taken approaches similar to that described in this paper. In both cases, however, graphic language translation ability was added to existing syntax directed compilers. This paper presents a translating scheme that is specifically addressed to translating a broad class of user oriented languages in a multi-computer graphic network. It is dedicated to the proposition that many specific application oriented languages are superior to one generalized language. It is based on the premise that the only things common among these many application oriented language systems are the lexical analyzer and a set of service routines. So it proposes a lexical analyzer in a remote processor and a set of service

routines in the main computer for the application programmer to use to implement his graphic interaction action.

ACKNOWLEDGMENTS

The author gratefully acknowledges the many people whose ideas have contributed to this paper. Chief among these contributors were A. Dean, M. Ling, and G. Link.

REFERENCES

- 1 I E SUTHERLAND
Sketchpad, a man-machine graphics communication system
SJCC vol 23 Spartan Books Inc Washington D C 1963
- 2 W H NINKE
Graphics I—a remote graphical display console system
Proceedings of the FJCC Las Vegas Nevada December 1965
- 3 D T ROSS
AED user kit
Massachusetts Institute of Technology, Electronic Systems Laboratory, various memoranda
- 4 E L JACKS
A laboratory for the study of graphical man-machine communication
Proceedings of the FJCC San Francisco California October 27-29 1964
- 5 R NARASIMHAN
Syntax-directed interpretation of classes of pictures
Comm. ACM, vol 9 no 3, March 1966
- 6 T S LING
General Electric reactive display system
Proceedings IEEE Region III Convention, Atlanta, Georgia, 1966
- 7 J W BACKUS
The syntax and semantics of the proposed international language of the Zurich ACM-GAMM conference ICIP
Paris, France, June 1959
- 8 CHOMSKY
On certain formal properties of grammars
Information & Control no 2 vol 2, 1959
- 9 T E CHEATHAM, JR.
Notes on compiling techniques
University of Michigan, Summer Conference Course on Automatic Programming, June 1966
- 10 L G ROBERTS
A graphical service system with variable syntax
Comm. ACM, vol 9 no 3, March 1966

Appendix A

Multishell statement definitions

General

<Multishell Program> ::= <Case Id. St.> *Send* <Structure Definition>
 <Structure Definition> ::= <Structure Drawing> <Property Definition>
 <Property Definition> ::= <Property Def. St.> | <Property Definition> <Property Def. St.>
 <Property Def. St.> ::= <Boundary Cond. St.> *Send* | <Member Prop. St.> *Send*
 <Structure Drawing> ::= <Structure St.> *Send* | <Structure Drawing> <Structure St.> *Send*

Case identification

<Case Id. St.> ::= *Multishell* <Id.> <Name> <Ext.> <Mail Drop> <Date> <Ref. RPM> <Ref. Temp.>
 <Id.> ::= <Long String>
 <Name> ::= <Long String>
 <Ext.> ::= <Short String>
 <Mail Drop> ::= <Short String>
 <Date> ::= <Long String>
 <Ref. RPM> ::= <Number>
 <Ref. Temp.> ::= <Number>

Structure drawing

<Structure St.> ::= *Draw Member* <Point Part> | <Structure St.> <Point Part>
 No more than two point parts
 <Point Part> ::= <Point Def.> | <Point Ref.>
 <Point Def.> ::= <Light Pen Coord. Pair> | <Typed Coord. Pair>
 <Erase St.> ::= *Erase* <Ref.>
 <Scale St.> ::= *Scale* <X-small> <X-large> <Y-small> <Y-large>
 <X-small> ::= <Number>
 <X-large> ::= <Number>
 <Y-small> ::= <Number>
 <Y-large> ::= <Number>

Boundary conditions

<Boundary Cond. St.> ::= *Joint* <Point Ref.> <Boundary Conditions>
 <Boundary Conditions> ::= <Radial Part> <Axial Part> <Rotational Part>
 <Radial Part> ::= <Radial Type> <Number> | <Null>
 <Axial Part> ::= <Axial Type> <Number> | <Null>
 <Rotational Part> ::= <Rotational Type> <Number> | <Null>
 <Radial Type> ::= $\uparrow$ | $\downarrow$ | Φ | Ψ
 <Axial Type> ::= $\rightarrow$ | $\leftarrow$ | $\otimes$ | $\odot$
 <Rotational Type> ::= M | $\bar{M}$ | R | $\bar{R}$

Member properties

<Member Prop. St.> ::= *Member* <Line Ref.> <Properties>
 <Properties> ::= <Material Prop.> <Thickness> <Rigidity> <Environment>
 <Material Prop.> ::= *M Data* <Number> | *Material Properties* <Poisson's Ratio>
 <Poisson's Ratio> ::= <Number>
 <Mass Density> ::= <Number>
 <I end E> ::= <Number>
 <I end A> ::= <Number>
 <J end E> ::= <Number>
 <J end A> ::= <Number>

<Thickness> ::= *Thickness* <I end Thickness> <J end Thickness>
 <I end Thickness> ::= <Number>
 <J end Thickness> ::= <Number>
 <Rigidity> ::= *Rigidity* <I end constraints> <J end constraints>
 <I end constraints> ::= ↑ | → | ↻ | Null
 <J end constraints> ::= ↑ | → | ↻ | Null
 <Environment> ::= *Environment* <Pressure> <Temp. I end> <Temp. J end>
 <Thermal Gradient>
 <Pressure> ::= <Number> | <Null>
 <Temp. I end> ::= <Number> | <Null>
 <Temp. J end> ::= <Number> | <Null>
 <Thermal Gradient> ::= <Number> | <Null>

Graphic language primitive definitions

Numbers

<Number> ::= <Unsigned decimal number> | <Adding Operator> <Unsigned decimal number>
 <Unsigned decimal number> ::= <Decimal Number> <Exponent Part>
 <Decimal Number> ::= <Unsigned Integer> . | <Unsigned Integer> | <Unsigned Integer> .
 <Unsigned Integer> | <Unsigned Integer>
 <Exponent Part> ::= <Null> | E <Integer>
 <Integer> ::= <Unsigned Integer> | <Adding Operator> <Unsigned Integer>
 <Unsigned Integer> ::= <Digit> | <Unsigned Integer> <Digit>
 Limit of 6 digits
 <Digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
 <Adding Operator> ::= + | -

Names

<Short String> ::= '<Basic String>'
 Limit of 6 Char.
 <Long String> ::= '<Basic String>'
 Limit of 24 Char.
 <Basic String> ::= <Char.> | <Basic String> <Char.>
 <Char.> ::= <Digit> | <Letter> | <Special Character>
 <Letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
 <Special Character> ::= SP|!|"|#|\$|%|&|(|)|*|+|,|-|.|/|:|;|<|=|>|?|@|[|N]|↑|←

Graphic primitives

<Typed Coord. Pair> ::= <Number> <Number>
 <Light Pen Coord. Pair> ::= *Light Pen Position* *Position*
 <Ref.> ::= <Point Ref.> | <Line Ref.> | <Item Ref.>
 <Point Ref.> ::= *Light Pen Point* *Hit*
 <Line Ref.> ::= *Light Pen Line* *Hit*
 <Item Ref.> ::= *Light Pen Item* *Hit*
 <Null> ::=